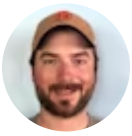
Everything you should know about certificates and PKI but are too afraid to ask

Updated on: May 20, 2024



Mike Malone

Follow Smallstep

Certificates and public key infrastructure (PKI) are hard. No shit, right? I know a lot of smart people who've avoided this particular rabbit hole. Personally, I avoided it for a long time and felt some shame for not knowing more. The obvious result was a vicious cycle: I was too embarrassed to ask questions so I never learned.

Eventually I was forced to learn this stuff because of what it enables: PKI lets you define a system cryptographically. It's universal and vendor neutral. It works everywhere so bits of your system can run anywhere and communicate securely. It's conceptually simple and super flexible. It lets you use TLS and ditch VPNs. You can ignore everything about your network and still have strong security characteristics. It's pretty great.

Now that I have learned, I regret not doing so sooner. PKI is really powerful, and really interesting. The math is complicated, and the standards are stupidly baroque, but the core concepts are actually quite simple. Certificates are the best way to identify code and devices, and identity is super useful for security, monitoring, metrics, and a million other things. Using certificates is not that hard. No harder than learning a new language or database. It's just slightly annoying and poorly documented.

This is the missing manual. I reckon most engineers can wrap their heads around all the most important concepts and common quirks in less than an hour. That's our goal here. An hour is a pretty small investment to learn something you literally can't do any other way.

My motives are mostly didactic. But I'll be using two open source projects we built at smallstep in various demonstrations: the step CLI and step certificates. To follow along you can `brew install step` to get both (see full install instructions here). If you prefer the easy button, spin up a free hosted authority using our Certificate Manager offering.

Let's start with a one sentence tl;dr: the goal of certificates and PKI is to bind names to public keys. That's it. The rest is just implementation details.

A broad overview and some words you should know

I'm going to use some technical terms, so let's go ahead and define them before we start.

An **entity** is anything that exists, even if it only exists logically or conceptually. Your computer is an entity. So is some code you wrote. So are you. So is the burrito you ate for lunch. So is the ghost that you saw when you were six -- even if your mom was right and it was just a figment of your imagination.

Every entity has an **identity**. This one's hard to define. Identity is what makes you you, ya know? On computers identity is usually represented as a bag of attributes describing some entity: group, age, location, favorite color, shoe size, whatever. An **identifier** is not the same as an identity. Rather, it's a unique reference to some entity that has an identity. I'm Mike, but Mike isn't my identity. It's a **name** -- identifier and name are synonyms (at least for our purposes).

Entities can **claim** that they have some particular name. Other entities might be able to authenticate that claim, confirming its truth. But a claim needn't be related to a name: I can make a claim about anything: my age, your age, access rights, the meaning of life, etc. **Authentication**, in general, is the process of confirming the truth of some claim.

A **subscriber** or **end entity** is an entity that's participating in a PKI and can be the **subject** of a certificate. A **certificate authority** (CA) is an entity that issues certificates to subscribers — a certificate **issuer**. Certificates that belong to subscribers are sometimes called end entity certificates or **leaf certificates** for reasons that'll become clearer once we discuss certificate chains. Certificates that belong to CAs are usually called **root certificates** or **intermediate certificates** depending on the sort of CA. Finally, a **relying party** is a certificate user that verifies and trusts certificates issued by a CA. To confuse matters a bit, an entity can be both a subscriber and a relying party. That is, a single entity can have its own certificate and use other certificates to authenticate remote peers (this is what happens with mutual TLS, for instance).

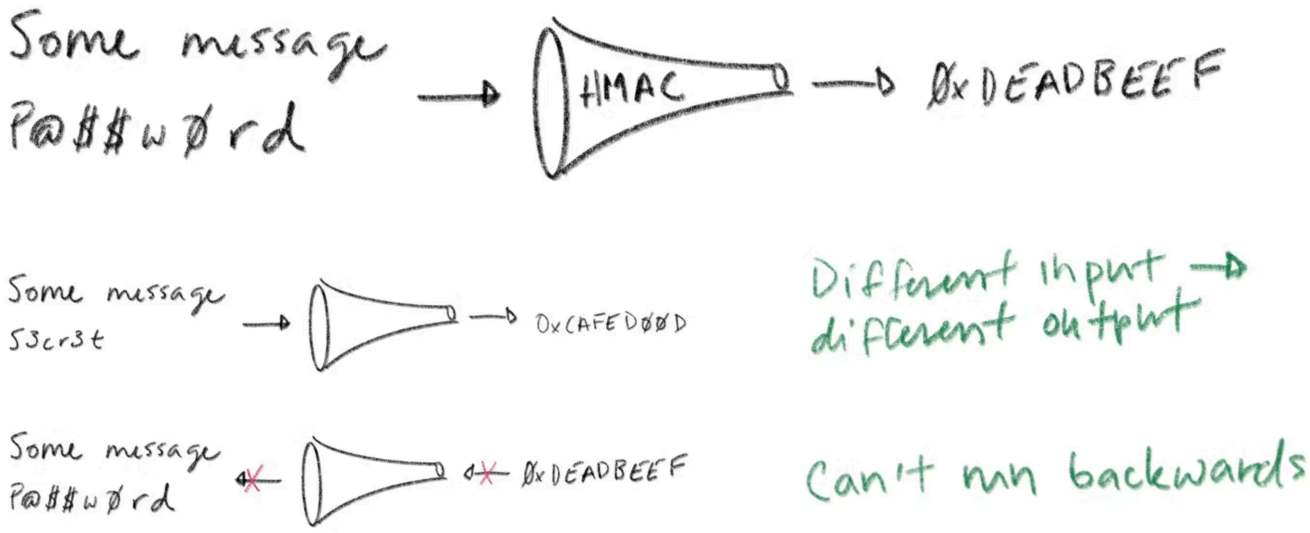
That's enough to get us started, but if pedagogy excites you consider putting RFC 4949 on your kindle. For everyone else, let's get concrete. How do we make claims and authenticate stuff in practice? Let's talk crypto.

MACs and signatures authenticate stuff

A **message authentication code** (MAC) is a bit of data that's used to verify which entity sent a message, and to ensure that a message hasn't been modified. The basic idea is to feed a shared secret (a password) along with a message through a hash function. The hash output is a MAC. You send the MAC along with the message to some recipient.

A recipient that also knows the shared secret can produce their own MAC and compare it to the one provided. Hash functions have a simple contract: if you feed them the same input twice you'll get the exact same output. If the input is different -- even by a single bit -- the output will be totally different. So if the recipient's MAC matches the one sent with the message it can be confident that the message was sent by another entity that knows the shared secret. Assuming only trusted entities know the shared secret, the recipient can trust the message.

Hash functions are also one-way: it's computationally infeasible to take the output of a hash function and reconstruct its input. This is critical to maintaining the confidentiality of a shared secret: otherwise some interloper could snoop your MACs, reverse your hash function, and figure out your secrets. That's no good. Whether this property holds depends critically on subtle details of how hash functions are used to build MACs. Subtle details that I'm not going to get into here. So beware: don't try to invent your own MAC algorithm. Use HMAC.



All this talk of MACs is prologue: our real story starts with *signatures*. A **signature** is conceptually similar to a MAC, but instead of using a shared secret you use a key pair (defined soon). With a MAC, at least two entities need to know the shared secret: the sender and the recipient. A valid MAC could have been generated by either party, and you can't tell which. Signatures are different. A signature can be verified using a public key but can only be generated with a corresponding private key. Thus, a recipient that only has a public key can verify signatures, but can't generate them. This gives you tighter control over who can sign stuff. If only one entity knows the private key you get a property called **non-repudiation**: the private key holder can't deny (repudiate) the fact that they signed some data.

If you're already confused, chill. They're called signatures for a reason: they're just like signatures in the real world. You have some stuff you want someone to agree to? You want to make sure you can prove they've agreed later on? Cool. Write it down and have them sign it.

Public key cryptography lets computers see

Certificates and PKI are built on **public key cryptography** (also called **asymmetric cryptography**), which uses **key pairs**. A key pair consists of a **public key** that can be distributed and shared with the world, and a corresponding **private key** that should be kept confidential by the owner.

Let's repeat that last part because it's important: the security of a public key cryptosystem depends on keeping private keys private.

There are two things you can do with a key pair:

- You can **encrypt** some data with the public key. The only way to decrypt that data is with the corresponding private key.
- You can **sign** some data with the private key. Anyone who knows the corresponding public key can verify the signature, proving which private key produced it.

Public key cryptography is a magical gift from mathematics to computer science. The math is complicated, for sure, but you don't need to understand it to appreciate its value. Public key cryptography lets computers do something that's otherwise impossible: **public key cryptography lets computers see**.

Ok, let me explain... public key cryptography lets one computer (or bit of code) prove to another that it knows something without sharing that knowledge directly. To prove you know a password you have to share it. Whoever you share it with can use it themselves. Not so with a private key. It's like vision. If you know what I look like you can tell who I am -- authenticate my identity -- by looking at me. But you can't shape-shift to impersonate me.

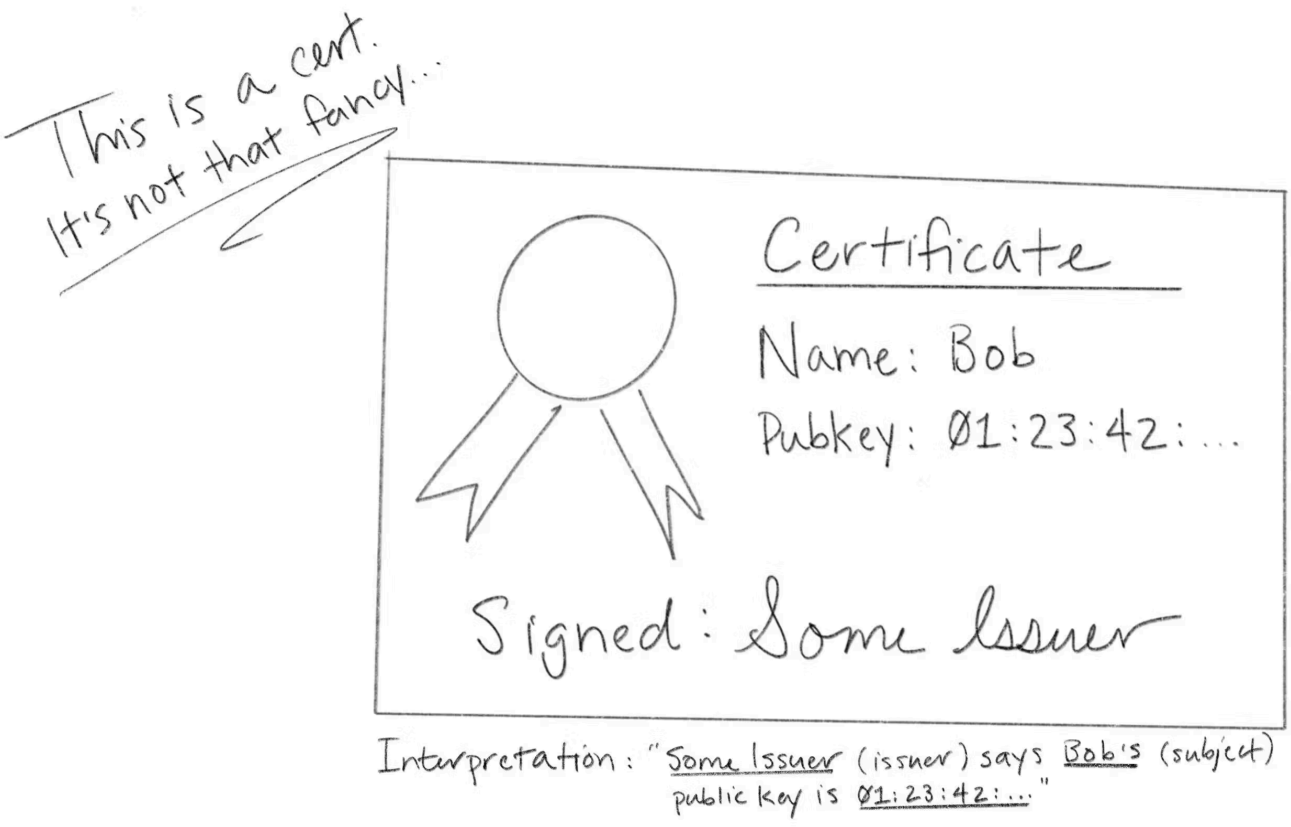
Public key cryptography does something similar. If you know my public key (what I look like) you can use it to see me across the network. You could send me a big random number, for example. I can sign your number and send you my signature. Verifying that signature is good evidence you're talking to me. This effectively allows computers to see who they're talking to across a network. This is so crazy useful we take it for granted in the real world. Across a network it's straight magic. Thanks math.

Certificates: driver's licenses for computers and code

What if you don't already know my public key? That's what certificates are for.

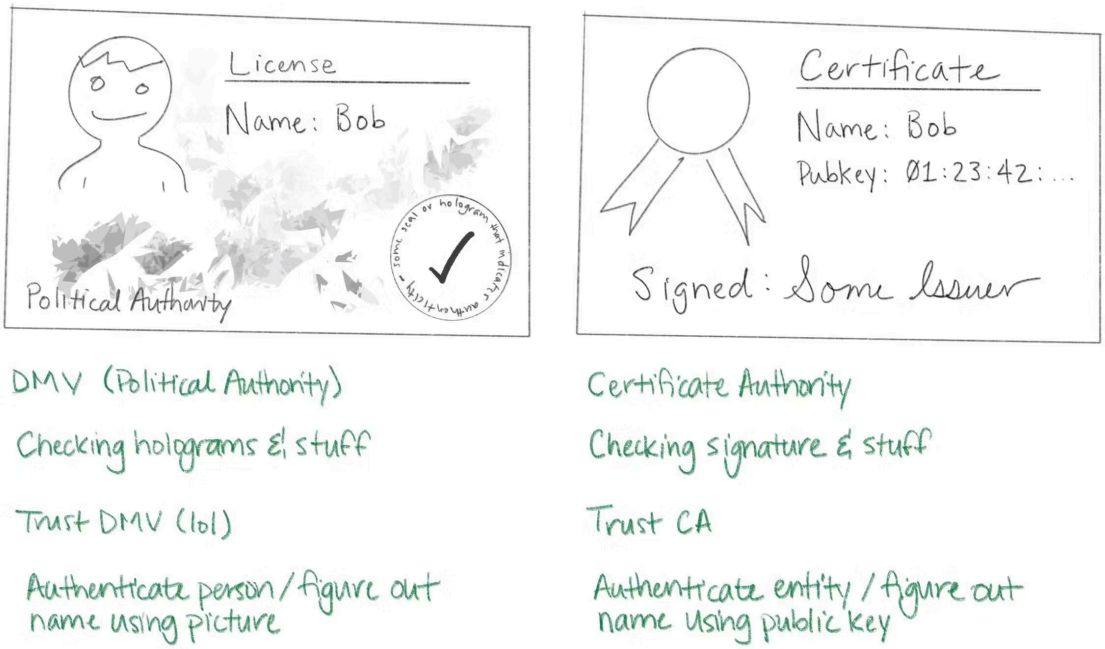
Certificates are fundamentally *really* simple. A certificate is a data structure that contains a public key and a name. The data structure is then *signed*. The signature *binds* the public key to the name. The entity that signs a certificate is called the **issuer** (or certificate authority) and the entity named in the certificate is called the **subject**.

If *Some Issuer* signs a certificate for *Bob*, that certificate can be interpreted as the statement: "*Some Issuer* says *Bob*'s public key is 01:23:42...".This is a claim made by *Some Issuer* about *Bob*. The claim is signed by *Some Issuer*, so if you know *Some Issuer*'s public key you can authenticate it by verifying the signature. If you trust *Some Issuer* you can trust the claim. Thus, certificates let you use trust, and knowledge of an issuer's public key, to learn another entity's public key (in this case, *Bob*'s). That's it. Fundamentally, that's all a certificate is.

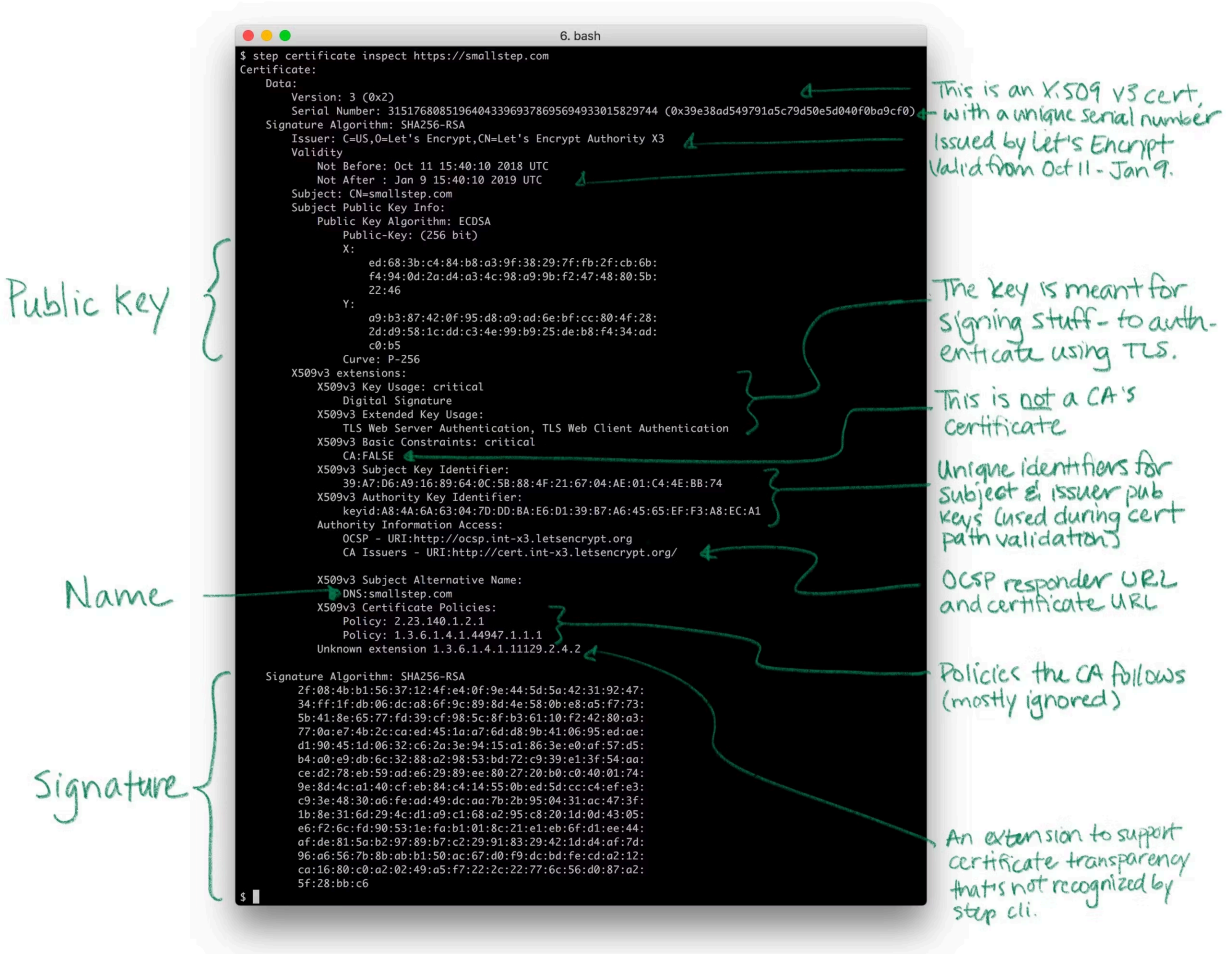


Certificates are like driver's licenses or passports for computers and code. If you've never met me before, but you trust the DMV, you can use my license for authentication:

verify that the license is valid (check hologram, etc), look at picture, look at me, read name. Computers use certificates to do the same thing: if you've never met some computer before, but you trust some certificate authority, you can use a certificate for authentication: verify that the certificate is valid (check signature, etc), look at public key, "look at private key" across network (as described above), read name.



Let's take a quick look at a real certificate:



Yea so I might have simplified the story a little bit. Like a driver's license, there's other stuff in certificates. Licenses say whether you're an organ donor and whether you're authorized to drive a commercial vehicle. Certificates say whether you're a CA and whether your public key is supposed to be used for signing or encryption. Both also have expirations.

There's a bunch of detail here, but it doesn't change what I said before: fundamentally, a certificate is just a thing that binds a public key to a name.

X.509, ASN.1, OIDs, DER, PEM, PKCS, oh my...

Let's look at how certificates are represented as bits and bytes. This part actually is annoyingly complicated. In fact, I suspect that the esoteric and poorly defined manner in which certificates and keys are encoded is the source of most confusion and frustration around PKI in general. This stuff is dumb. Sorry.

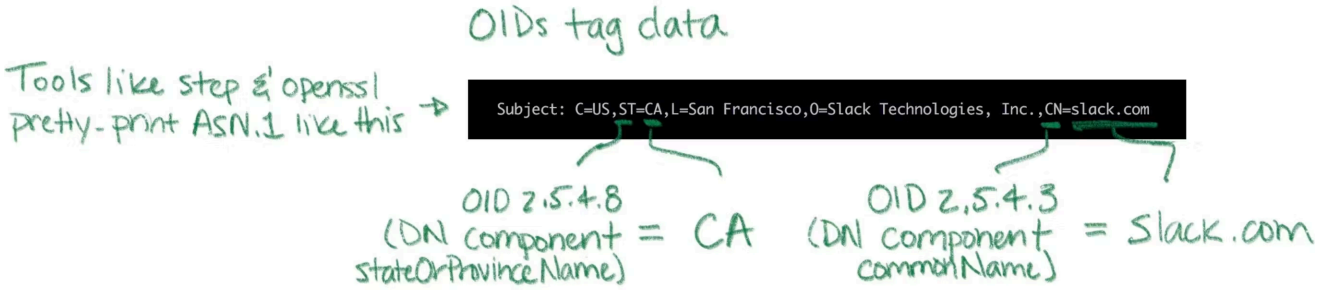
Usually when people talk about certificates without additional qualification they're referring to X.509 v3 certificates. More specifically, they're usually talking about the PKIX variant described in RFC 5280 and further refined by the CA/Browser Forum's Baseline Requirements. In other words, they're referring to the sort of certificates that browsers understand and use for HTTPS (HTTP over TLS). There are other certificate formats. Notably, SSH and PGP both have their own. But we're going to focus on X.509. If you can understand X.509 you'll be able to figure everything else out.

Since these certificates are so broadly supported -- they have good libraries and whatnot -- they're frequently used in other contexts, too. They're certainly the most common format for certificates issued by internal PKI (defined in a bit). Importantly, these certificates work out of the box with TLS and HTTPS clients and servers.

You can't fully appreciate X.509 without a small history lesson. X.509 was first standardized in 1988 as part of the broader X.500 project under the auspices of the ITU-T (the International Telecommunications Union's standards body). X.500 was an effort by the telcos to build a global telephone book. That never happened, but vestiges remain. If you've ever looked at an X.509 certificate and wondered why something designed for the web encodes a locality, state, and country here's your answer: X.509 wasn't designed for the web. It was designed thirty years ago to build a phone book.

X.509 builds on ASN.1, another ITU-T standard (defined by X.208 and X.680). ASN stands for Abstract Syntax Notation (1 stands for One). ASN.1 is a notation for defining data types. You can think of it like JSON for X.509 but it's actually more like protobuf or thrift or SQL DDL. RFC 5280 uses ASN.1 to define an X.509 certificate as an object that contains various bits of information: a name, key, signature, etc.

ASN.1 has normal data types like integers, strings, sets, and sequences. It also has an unusual type that's important to understand: object identifiers (OIDs). An OID is like a URI, but more annoying. They're (supposed to be) universally unique identifiers. Structurally, OIDs are a sequence of integers in a hierarchical namespace. You can use an OID to *tag* a bit of data with a type. A string is just a string, but if I tag a string with OID `2.5.4.3` then it's no longer an ordinary string -- it's an X.509 *common name*.



ASN.1 is *abstract* in the sense that the standard doesn't say anything about how stuff should be represented as bits and bytes. For that there are various *encoding rules* that specify concrete representations for ASN.1 data values. It's an additional abstraction layer that's supposed to be useful, but is mostly just annoying. It's sort of like the difference between unicode and utf8 (eek).

There are a bunch of encoding rules for ASN.1, but there's only one that's commonly used for X.509 certificates and other crypto stuff: distinguished encoding rules or DER (though the non-canonical basic encoding rules (BER) are also occasionally used). DER is a pretty simple type-length-value encoding, but you really don't need to worry about it since libraries will do most of the heavy lifting.

Unfortunately, the story doesn't stop here. You don't have to worry much about encoding and decoding DER but you *definitely will* need to figure out whether a particular certificate is a plain DER-encoded X.509 certificate or something fancier. There are two potential dimensions of fanciness: we might be looking at something more than raw DER, and we might be looking at something more than just a certificate.

Starting with the former dimension, DER is straight binary, and binary data is hard to copy-paste and otherwise shunt around the web. So most certificates are packaged up in PEM files (which stands for *Privacy Enhanced EMail*, another weird historical vestige). If you've ever worked with MIME, PEM is similar: a base64 encoded payload sandwiched between a header and a footer. The PEM header has a label that's supposed to describe the payload. Shockingly, this simple job is mostly botched and PEM labels are often inconsistent between tools (RFC 7468 attempts to standardize the use of PEM in this context, but it's not complete and not always followed). Without further ado, here's what a PEM-encoded X.509 v3 certificate looks like:

```
-----BEGIN CERTIFICATE-----
MIIBwzCCAWqgAwIBAgIRAIi5QRl9kz1wb+SUP20gB1kwCgYIKoZIzj0EAwIwGzEZ
MBcGA1UEAxMQTDVkIFRlc3QgUm9vdCBDQTAeFw0xODExMDYyMjA0MDNaFw0yODEx
MDMyMjA0MDNaMCMxITAfBgNVBAMTGEw1ZCBUZXN0IEludGVybWVkaWF0ZSBDQTBZ
MBMGByqGSM49AgEGCCqGSM49AwEHA0IABAST8h+JftPkPocZyuZ5CVuPUk3vUtgo
cgRbkYk7Ong7ey/fM5fJdRNdeW6SouV5h3nF9JvYKEXuoymSNjGbKomjgYYwgYMw
DgYDVR0PAQH/BAQDAgGmMB0GA1UdJQQWMBQGCCsGAQUFBwMBBggrBgEFBQcDAjAS
BgNVHRMBAf8ECDAGAQH/AgEAMB0GA1UdDgQWBBRc+LHppFk8sflIpm/XKpbNMwx3
SDAfBgNVHSMEGDAWgBTirEpzC7/gexnnz7ozjWKd71lz5DAKBggqhkjOPQQDAgNH
ADBEAiAejDEfua7dud78lxWe9eYxYcM93mlUMFIzbWlOJzg+rgIgcdtU9wIKmn5q
FU3iOiRP5VyLNmrsQD3/ItjUN1f1ouY=
-----END CERTIFICATE-----
```

PEM-encoded certificates will usually carry a `.pem`, `.crt`, or `.cer` extension. A raw certificate encoded using DER will usually carry a `.der` extension. Again, there's not much consistency here, so your mileage may vary.

Returning to our other dimension of fanciness: in addition to fancier encoding using PEM, a certificate might be wrapped up in fancier packaging. Several *envelope formats* define larger data structures (still using ASN.1) that can contain certificates, keys, and other stuff. Some things ask for "a certificate" when they really want a certificate in one of these envelopes. So beware.

The envelope formats you're likely to encounter are part of a suite of standards called PKCS (Public Key Cryptography Standards) published by RSA labs (actually the story is slightly more complicated, but whatever). The first is PKCS#7, rebranded Cryptographic Message Syntax (CMS) by IETF, which can contain one or more certificates (encoding a full certificate chain, described shortly). PKCS#7 is commonly used by Java. Common extensions are `.p7b` and `.p7c`. The other common envelope format is PKCS#12 which can contain a certificate chain (like PKCS#7) along with an (encrypted) private key. PKCS#12 is commonly used by Microsoft products. Common extensions are `.pfx` and `.p12`. Again, the PKCS#7 and PKCS#12 envelopes also use ASN.1. That means both can be encoded as raw DER or BER or PEM. That said, in my experience they're almost always raw DER.

Key encoding is similarly convoluted, but the pattern is generally the same: some ASN.1 data structure describes the key, DER is used as a binary encoding, and PEM (hopefully with a useful header) might be used as a slightly friendlier representation. Deciphering

the sort of key you're looking at is half art, half science. If you're lucky RFC 7468 will give good guidance to figure out what your PEM payload is. Elliptic curve keys are usually labeled as such, though there doesn't seem to be any standardization. Other keys are simply "PRIVATE KEY" by PEM. This usually indicates a PKCS#8 payload, an envelope for private keys that includes key type and other metadata. Here's an example of a PEM-encoded elliptic curve key:

```
$ step crypto keypair --kty EC --no-password --insecure ec.pub ec.prv
$ cat ec.pub ec.prv
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEc73/+JOESKlqWlhf0UzcRjEe7inF
uu2z1DWxr+2YRLfTaJOm9huerJCh71z5lugg+QVLZBedKGEff5jgTssXHg==
-----END PUBLIC KEY-----
-----BEGIN EC PRIVATE KEY-----
MHcCAQEEICjpa3i7ICHSIqZPZfkJpcRim/EAmUtMFGJg6QjkMqDMoAoGCCqGSM49
AwEHoUQDQgAEc73/+JOESKlqWlhf0UzcRjEe7inFuu2z1DWxr+2YRLfTaJOm9hue
rJCh71z5lugg+QVLZBedKGEff5jgTssXHg==
-----END EC PRIVATE KEY-----
```

It's also quite common to see private keys encrypted using a password (a shared secret or symmetric key). Those will look something like this (`Proc-Type` and `DEK-Info` are part of PEM and indicate that this PEM payload is encrypted using `AES-256-CBC`):

```
-----BEGIN EC PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-256-CBC,b3fd6578bf18d12a76c98bda947c4ac9

qdV5u+wrywkbO0Ai8VUuwZO1cqhwsNaDQwTiYUwohvot7Vw851rW/43poPhH07So
sdLFVCKPd9v6F9n2dkdWCeeFlI4hfx+EwzXLuaRWg6aoYOj7ucJdkofyRyd4pEt+
Mj60xqLkaRtphh9HWKgaHsdBki68LQbObLOz4c6SyxI=
-----END EC PRIVATE KEY-----
```

PKCS#8 objects can also be encrypted, in which case the header label should be "ENCRYPTED PRIVATE KEY" per RFC 7468. You won't have `Proc-Type` and `Dek-Info`

headers in this case as this information is encoded in the payload instead.

Public keys will usually have a `.pub` or `.pem` extension. Private keys may carry a `.prv`, `.key`, or `.pem` extension. Once again, your mileage may vary.

Quick summary. ASN.1 is used to define data types like certificates and keys. DER is a set of encoding rules for turning ASN.1 into bits and bytes. X.509 is defined in ASN.1. PKCS#7 and PKCS#12 are bigger data structures, also defined using ASN.1, that can contain certificates and other stuff. They're commonly used by Java and Microsoft, respectively. Since raw binary DER is hard to shunt around the web most certificates are PEM-encoded, which base64 encodes the DER and labels it. Private keys are usually represented as PEM-encoded PKCS#8 objects. Sometimes they're also encrypted with a password.

If that's confusing, it's not you. It's the world. I tried.

PUBLIC KEY INFRASTRUCTURE

It's good to know what a certificate is, but that's less than half the story. Let's look at how certificates are created and used.

Public key infrastructure (PKI) is the umbrella term for all of the stuff we need in order to issue, distribute, store, use, verify, revoke, and otherwise manage and interact with certificates and keys. It's an intentionally vague term, like "database infrastructure". Certificates are the building blocks of most PKIs, and certificate authorities are the foundation. That said, PKI is so much more. It includes libraries, cron jobs, protocols, conventions, clients, servers, people, processes, names, discovery mechanisms, and all the other stuff you'll need to use public key cryptography effectively.

If you build your own PKI from scratch you'll enjoy a ton of discretion. Just like if you build your own database infrastructure. In fact, many simple PKIs don't even use certificates. When you edit `~/.ssh/authorized_keys` you're configuring a simple certificate-less form of PKI that SSH uses to bind public keys to names in flat files. PGP uses certificates, but doesn't use CAs. Instead it uses a web-of-trust model. You can even use a blockchain to assign names and bind them to public keys. The only thing that's truly mandatory if you're building a PKI from scratch is that, definitionally, you've got to be using public keys. Everything else can change.

That said, you probably don't want to build a PKI entirely from scratch. We'll focus on the sort of PKI used on the web, and internal PKIs that are based on Web PKI technologies and leverage existing standards and components.

As we proceed remember the simple goal of certificates and PKI: to bind names to public keys.

WEB PKI VS INTERNAL PKI

You interact with Web PKI via your browser whenever you access an HTTPS URL — like when you loaded this website. This is the only PKI many people are (at least vaguely) familiar with. It creaks and clanks and bumbles along but it mostly works. Despite its problems, it substantially improves security on the web and it's mostly transparent to users. You should use it everywhere your system communicates with the outside world over the internet.

Web PKI is mostly defined by RFC 5280 and refined by the CA/Browser Forum (a.k.a., CA/B or CAB Forum). It's sometimes called "Internet PKI" or PKIX (after the working group that created it). The PKIX and CAB Forum documents cover a lot of ground. They define the variety of certificates we talked about in the last section. They also define what a "name" is and where it goes in a certificate, what signature algorithms can be used, how a relying party determines the issuer of a certificate, how a certificate's validity period (issue and expiry dates) is specified, how revocation and certificate path validation works, the process that CAs use to determine whether someone owns a domain, and a whole lot more.

Web PKI is important because Web PKI certificates work by default with browsers and pretty much everything else that uses TLS.

Internal PKI is PKI you run yourself, for your own stuff: production infrastructure like services, containers, and VMs; enterprise IT applications; corporate endpoints like laptops and phones; and any other code or device you want to identify. It allows you to authenticate and establish cryptographic channels so your stuff can run anywhere and securely communicate, even across the public internet.

Why run your own internal PKI if Web PKI already exists? The simple answer is that Web PKI wasn't designed to support internal use cases. Even with a CA like Let's Encrypt, which offers free certificates and automated provisioning, you'll have to deal with rate

limits and availability. That's no good if you have lots of services that you deploy all the time.

Further, with Web PKI you have little or no control over important details like certificate lifetime, revocation mechanisms, renewal processes, key types, and algorithms (all important stuff we'll explain in a moment).

Finally, the CA/Browser Forum Baseline Requirements actually prohibit Web PKI CAs from binding internal IPs (e.g., stuff in `10.0.0.0/8`) or internal DNS names that aren't fully-qualified and resolvable in public global DNS (e.g., you can't bind a kubernetes cluster DNS name like `foo.ns.svc.cluster.local`). If you want to bind this sort of name in a certificate, issue lots of certificates, or control certificate details, you'll need your own internal PKI.

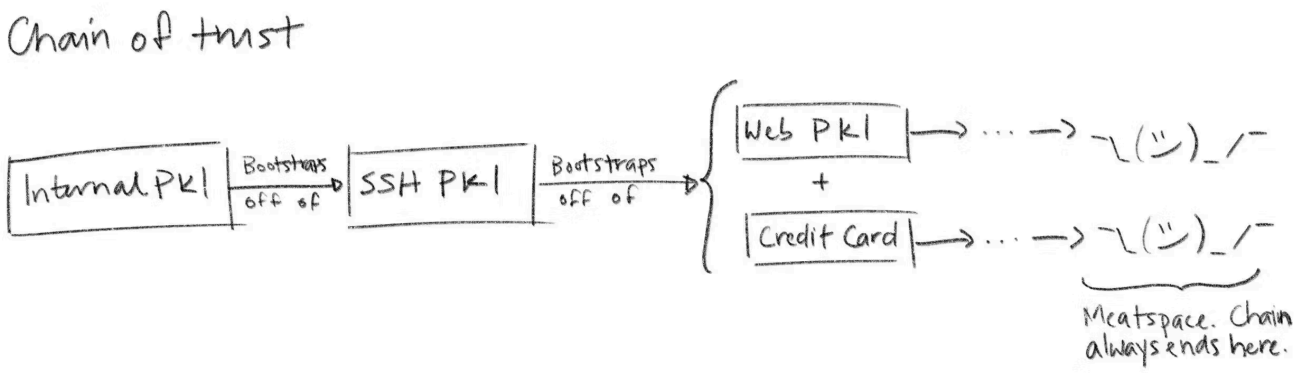
In the next section we'll see that trust (or lack thereof) is yet *another* reason to avoid Web PKI for internal use. In short, use Web PKI for your public website and APIs. Use your own internal PKI for everything else.

TRUST & TRUSTWORTHINESS

Trust Stores

Earlier we learned to interpret a certificate as a statement, or claim, like: "*issuer says subject's public key is blah blah blah*". This claim is signed by the issuer so it can be authenticated by relying parties. We glossed over something important in this description: "how does the relying party know the *issuer*'s public key?

The answer is simple, if not satisfying: relying parties are pre-configured with a list of trusted **root certificates** (or trust anchors) in a **trust store**. The manner in which this pre-configuration occurs is an important aspect of any PKI. One option is to bootstrap off of another PKI: you could have some automation tool use SSH to copy root certificates to relying parties, leveraging the SSH PKI described earlier. If you're running in the cloud your SSH PKI, in turn, is bootstrapped off of Web PKI plus whatever authentication your cloud vendor did when you created your account and gave them your credit card. If you follow this *chain of trust* back far enough you'll always find people: every trust chain ends in meatspace.



Root certificates in trust stores are **self-signed**. The issuer and the subject are the same. Logically it's a statement like "*Mike* says *Mike*'s public key is *blah blah blah*". The signature on a self-signed certificate provides assurance that the subject/issuer knows the relevant private key, but anyone can generate a self-signed certificate with any name they want in it. So provenance is critical: a self-signed certificate should only be trusted insofar as the process by which it made its way into the trust store is trusted. On macOS the trust store is managed by the keychain. On many Linux distributions it's simply some file(s) in `/etc` or elsewhere on disk. If your users can modify these files, you better trust all your users.

So where do trust stores come from? For Web PKI the most important relying parties are web browsers. The trust stores used by default by the major browsers -- and pretty much everything else that uses TLS -- are maintained by four organizations:

- Apple's root certificate program used by iOS and macOS
- Microsoft's root certificate program used by Windows
- Mozilla's root certificate program used by their products and, because of its open and transparent process, used as the basis for many other trust stores (e.g., for

many Linux distributions)

- Google's root certificate program used by Chrome on all platforms except iOS.

Operating system trust stores typically ship with the OS. Firefox ships with its own trust store (distributed using TLS from mozilla.org — bootstrapping off of Web PKI using some other trust store). Programming languages and other non-browser stuff like `curl` typically use the OS trust store by default. So the trust stores typically used by default by pretty much everything come pre-installed and are updated via software updates (which are usually code signed using yet another PKI).

There are more than 100 certificate authorities commonly included in the trust stores maintained by these programs. You probably know the big ones: Let's Encrypt, Symantec, DigiCert, Entrust, etc. It can be interesting to peruse them. If you'd like to do so programmatically, Cloudflare's cfssl project maintains a github repository that includes the trusted certificates from various trust stores to assist with certificate bundling (which we'll discuss momentarily). For a more human-friendly experience you can query Censys to see which certificates are trusted by Mozilla, Apple, and Microsoft.

Trustworthiness

These 100+ certificate authorities are trusted in the descriptive sense — browsers and other stuff trust certificates issued by these CAs by default. But that doesn't mean they're *trustworthy* in the moral sense. On the contrary, there are documented cases of Web PKI certificate authorities providing governments with fraudulent certificates in order to snoop on traffic and impersonate websites. Some of these "trusted" CAs operate out of authoritarian jurisdictions like China. Democracies don't really have a moral high ground here, either. NSA takes every available opportunity to undermine Web PKI. In 2011 the "trusted" DigiNotar and Comodo certificate authorities were both compromised. The DigiNotar breach was probably NSA. There are also numerous examples of CAs mistakenly issuing malformed or non-compliant certificates. So while these CAs are de-facto trusted, as a group they're empirically *not* trustworthy. We'll soon see that Web PKI in general is only as secure as the least secure CA, so this is not good.

The browser community has taken some action to address this issue. The CA/Browser Forum Baseline Requirements rationalize the rules that these trusted certificate authorities are supposed to follow before issuing certificates. CAs are audited for compliance with these rules as part of the WebTrust audit program, which is required by some root certificate programs for inclusion in their trust stores (e.g., Mozilla's).

Still, if you're using TLS for internal stuff, you probably don't want to trust these public CAs any more than you have to. If you do, you're probably opening up the door to NSA and others. You're accepting the fact that your security depends on the discipline and scruples of 100+ other organizations. Maybe you don't care, but fair warning.

Federation

To make matters worse, Web PKI relying parties (RPs) trust every CA in their trust store to sign certificates for any subscriber. The result is that the overall security of Web PKI is only as good as the least secure Web PKI CA. The 2011 DigiNotar attack demonstrated the problem here: as part of the attack a certificate was fraudulently issued for google.com. This certificate was trusted by major web browsers and operating systems despite the fact that Google had no relationship with DigiNotar. Dozens more fraudulent certificates were issued for companies like Yahoo!, Mozilla, and The Tor Project. DigiNotar root certificates were ultimately removed from the major trust stores, but a lot of damage had almost certainly already been done.

More recently, Sennheiser got called out for installing a self-signed root certificate in trust stores with their HeadSetup app, then embedding the corresponding private key in the app's configuration. Anyone can extract this private key and use it to issue a certificate for any domain. Any computer that has the Sennheiser certificate in its trust store would trust these fraudulent certificates. This completely undermines TLS. Oops.

There are a number of mitigation mechanisms that can help reduce these risks. Certificate Authority Authorization (CAA) allows you to restrict which CAs can issue certificates for your domain using a special DNS record. Certificate Transparency (CT) (RFC 6962) mandates that CAs submit every certificate they issue to an impartial observer that maintains a public certificate log to detect fraudulently issued certificates. Cryptographic proof of CT submission is included in issued certificates. HTTP Public Key Pinning (HPKP or just "pinning") lets a subscriber (a website) tell an RP (a browser) to only accept certain public keys in certificates for a particular domain.

The problem with all of these things is RP support, or lack thereof. The CAB Forum now mandates CAA checks in browsers. Some browsers also have some support for CT and HPKP. For other RPs (e.g., most TLS standard library implementations) this stuff is almost never enforced. This issue will come up repeatedly: a lot of certificate policy must be enforced by RPs, and RPs can rarely be bothered. If RPs don't check CAA records and don't require proof of CT submission this stuff doesn't do much good.

In any case, if you run your own internal PKI you should maintain a separate trust store for internal stuff. That is, instead of adding your root certificate(s) to the existing system trust store, configure internal TLS requests to use only your roots. If you want better federation internally (e.g., you want to restrict which certificates your internal CAs can issue) you might try CAA records and properly configured RPs. You might also want to check out SPIFFE, an evolving standardization effort that addresses this problem and a number of others related to internal PKI.

WHAT'S A CERTIFICATE AUTHORITY

We've talked a lot about certificate authorities (CAs) but haven't actually defined what one is. A CA is a trusted certificate issuer. It vouches for the binding between a public key and a name by signing a certificate. Fundamentally, a certificate authority is just another certificate and a corresponding private key that's used to sign other certificates.

Obviously some logic and process needs to be wrapped around these artifacts. The CA needs to get its certificate distributed in trust stores, accept and process certificate requests, and issue certificates to subscribers. A CA that exposes remotely accessible APIs to automate this stuff it's called an *online CA*. A CA with a self-signed root certificate included in trust stores is called a *root CA*.

Intermediates, Chains, and Bundling

The CAB Forum Baseline Requirements stipulate that a root private key belonging to a Web PKI root CA can only be used to sign a certificate by issuing a direct command (see section 4.3.1). In other words, Web PKI root CAs can't automate certificate signing. They can't be online. This is a problem for any large scale CA operation. You can't have someone manually type a command into a machine to fulfill every certificate order.

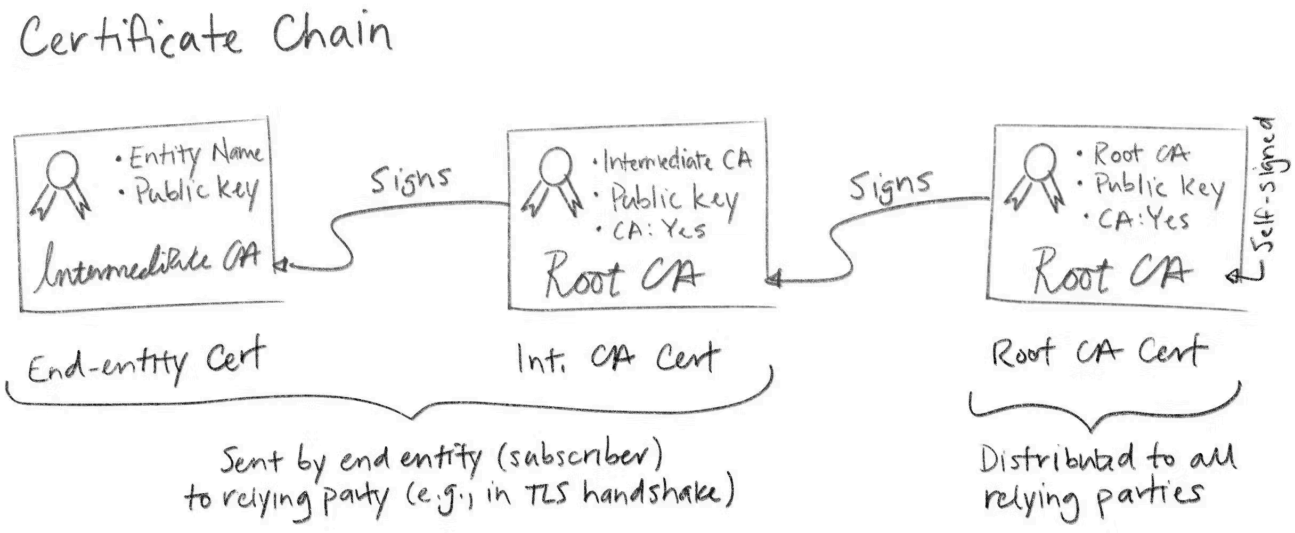
The reason for this stipulation is security. Web PKI root certificates are broadly distributed in trust stores and hard to revoke. Compromising a root CA private key would affect literally billions of people and devices. Best practice, therefore, is to keep root private keys offline, ideally on some specialized hardware connected to an air gapped machine, with good physical security, and with strictly enforced procedures for use.

Many internal PKIs also follow these same practices, though it's far less necessary. If you can automate root certificate rotation (e.g., update your trust stores using

configuration management or orchestration tools) you can easily rotate a compromised root key. People obsess so much over root private key management for internal PKIs that it delays or prevents internal PKI deployment. Your AWS root account credentials are at least as sensitive, if not more. How do you manage those credentials?

To make certificate issuance scalable (i.e., to make automation possible) when the root CA isn't online, the root private key is only used infrequently to sign a few *intermediate certificates*. The corresponding *intermediate private keys* are used by intermediate CAs (also called subordinate CAs) to sign and issue leaf certificates to subscribers. Intermediates aren't generally included in trust stores, making them easier to revoke and rotate, so certificate issuance from an intermediate typically is online and automated.

This **bundle** of certificates -- leaf, intermediate, root -- forms a chain (called a *certificate chain*). The leaf is signed by the intermediate, the intermediate is signed by the root, and the root signs itself.



Technically this is another simplification. There's nothing stopping you from creating longer chains and more complex graphs (e.g., by cross-certification). This is generally discouraged though, as it can become very complicated very quickly. In any case, end entity certificates are leaf nodes in this graph. Hence the name "leaf certificate".

When you configure a subscriber (e.g., a web server like Apache or Nginx or Linkerd or Envoy) you'll typically need to provide not just the leaf certificate, but a certificate bundle that includes intermediate(s). PKCS#7 and PKCS#12 are sometimes used here because they can include a full certificate chain. More often, certificate chains are

encoded as a simple sequence of line-separated PEM objects. Some stuff expects the certs to be ordered from leaf to root, other stuff expects root to leaf, and some stuff doesn't care. More annoying inconsistency. Google and Stack Overflow help here. Or trial and error.

In any case, here's an example:

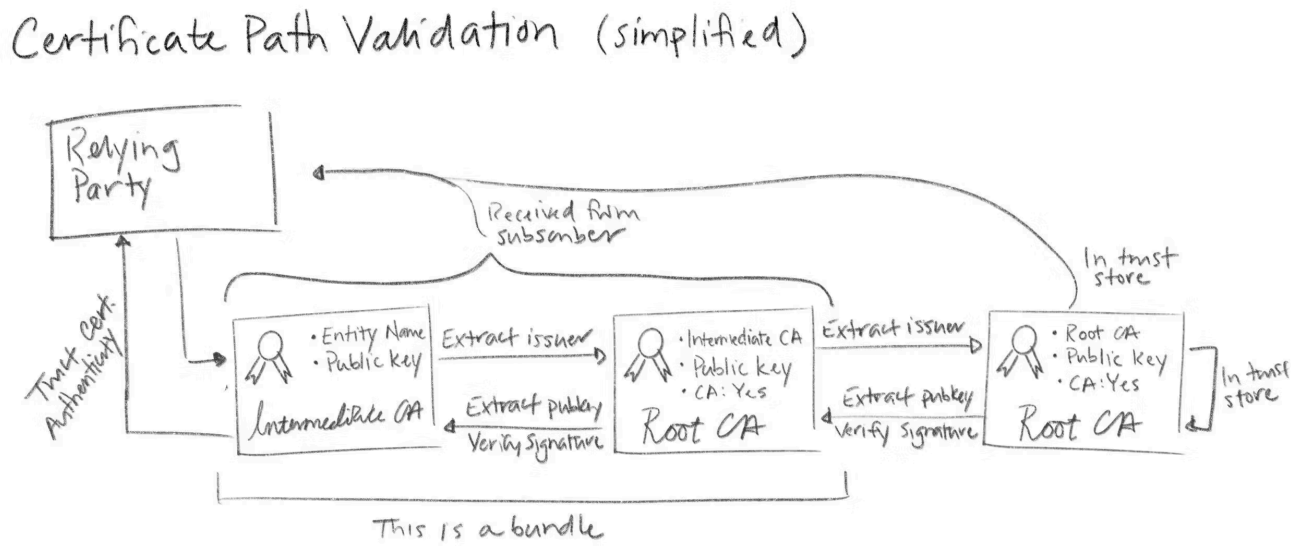
```
$ cat server.crt
-----BEGIN CERTIFICATE-----
MIICFDCCAbmgAwIBAgIRANE187UXf5fn5TgXSq65CMQwCgYIKoZIzj0EAwIwHzEd
MBsGA1UEAxMUVGVzdCBJbnRlcm1lZGlhdGUgQ0EwHhcNMTgxMjA1MTc0OTQ0WhcN
MTgxMjA2MTc0OTQ0WjAUMRIwEAYDVQQDEwlsb2NhbGhvc3QwWTATBgcqhkjOPQIB
BggqhkjOPQMBBwNCAAQqE2VPZ+uS5q/XiZd6x6vZSKAYFM4xrYa/ANmXeZ/gh/n0
vhsmXIKNCg6vZh69FCbBMZdYEVOb7BRQIR8Q1qjGo4HgMIHdMA4GA1UdDwEB/wQE
AwIFoDAdBgNVHSUEFjAUBggrBgEFBQcDAQYIKwYBBQUHAwIwHQYDVR0OBBYEFHee
8N698LZWzJg6SQ9F6/gQBGkmMB8GA1UdIwQYMBaAFAZ0jCINuRtVd6ztucMf8Bun
D++sMBQGA1UdEQQNMAuCCWxvY2FsaG9zdDBWBgwrBgEEAYKkZMYoQAEERjBEAgEB
BBJtaWtlQHNtYWxsc3RlcC5jb20EK0lxOWItOEdEUWg1SmxZaUJwSTBBRW01eHN5
YzM0d0dNUkJWRXE4ck5pQzQwCgYIKoZIzj0EAwIDSQAwRgIhAPL4SgbHIbLwfRqO
HO3iTsozZsCuqA34HMaqXveiEie4AiEAhUjjb7vCGuPpTmn8HenA5hJplr+Ql8s1
d+SmYsT0jDU=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIBuzCCAWKgAwIBAgIRAKBv/7Xs6GPAK4Y8z4udSbswCgYIKoZIzj0EAwIwFzEV
MBMGA1UEAxMMVGVzdCBSb290IENBMB4XDTE4MTIwNTE3MzgzOFoXDTI4MTIwMjE3
MzgzOFowHzEdMBsGA1UEAxMUVGVzdCBJbnRlcm1lZGlhdGUgQ0EwWTATBgcqhkjO
PQIBBggqhkjOPQMBBwNCAAT8r2WCVhPGeh2J2EFdmdMQi5YhpMp3hyVZWu6XNDbn
xd8QBUNZTHqdsMKDtXoNfmhH//dwz78/kRnbka+acJQ9o4GGMIGDMA4GA1UdDwEB
/wQEAwIBpjAdBgNVHSUEFjAUBggrBgEFBQcDAQYIKwYBBQUHAwIwEgYDVR0TAQH/
BAgwBgEB/wIBADAdBgNVHQ4EFgQUBnSMIg25G1V3rO25wx/wG6cP76wwHwYDVR0j
BBgwFoAUcITNjk2XmInW+xfLJjMYVMG7fMswCgYIKoZIzj0EAwIDRwAwRAIgTCgI
BRvPAJZb+soYP0tnObqWdplmO+krWmHqCWtK8hcCIHS/es7GBEj3bmGMus+8n4Q1
x8YmK7ASLmSCffCTct9Y
-----END CERTIFICATE-----
```

Again, annoying and baroque, but not rocket science.

Certificate path validation

Since intermediate certificates are not included in trust stores they need to be distributed and verified just like leaf certificates. You provide these intermediates when

you configure subscribers, as described above. Then subscribers pass them along to RPs. With TLS this happens as part of the handshake that establishes a TLS connection. When a subscriber sends its certificate to a relying party it includes any intermediate(s) necessary to chain back up to a trusted root. The relying party verifies the leaf and intermediate certificates in a process called **certificate path validation**.



The complete certificate path validation algorithm is complicated. It includes checking certificate expirations, revocation status, various certificate policies, key use restrictions, and a bunch of other stuff. Proper implementation of this algorithm by PKI RPs is absolutely critical. People are shockingly casual about disabling certificate path validation (e.g., by passing the `-k` flag to `curl`). Don't do this.

Don't disable certificate path validation. It's not that hard to do proper TLS, and certificate path validation is the part of TLS that does authentication. People sometimes argue that the channel is still encrypted, so it doesn't matter. That's wrong. It does matter. Encryption without authentication is pretty worthless. It's like a blind confessional: your conversation is private but you have no idea who's on the other side of the curtain. Only this isn't a church, it's the internet. So **don't disable certificate path validation**.

KEY & CERTIFICATE LIFECYCLE

Before you can use a certificate with a protocol like TLS you need to figure out how to get one from a CA. Abstractly this is a pretty simple process: a subscriber that wants a certificate generates a key pair and submits a request to a certificate authority. The CA

makes sure the name that will be bound in the certificate is correct and, if it is, signs and returns a certificate.

Certificates expire, at which point they're no longer trusted by RPs. If you're still using a certificate that's about to expire you'll need to renew and rotate it. If you want RPs to stop trusting a certificate before it expires, it can (sometimes) be revoked.

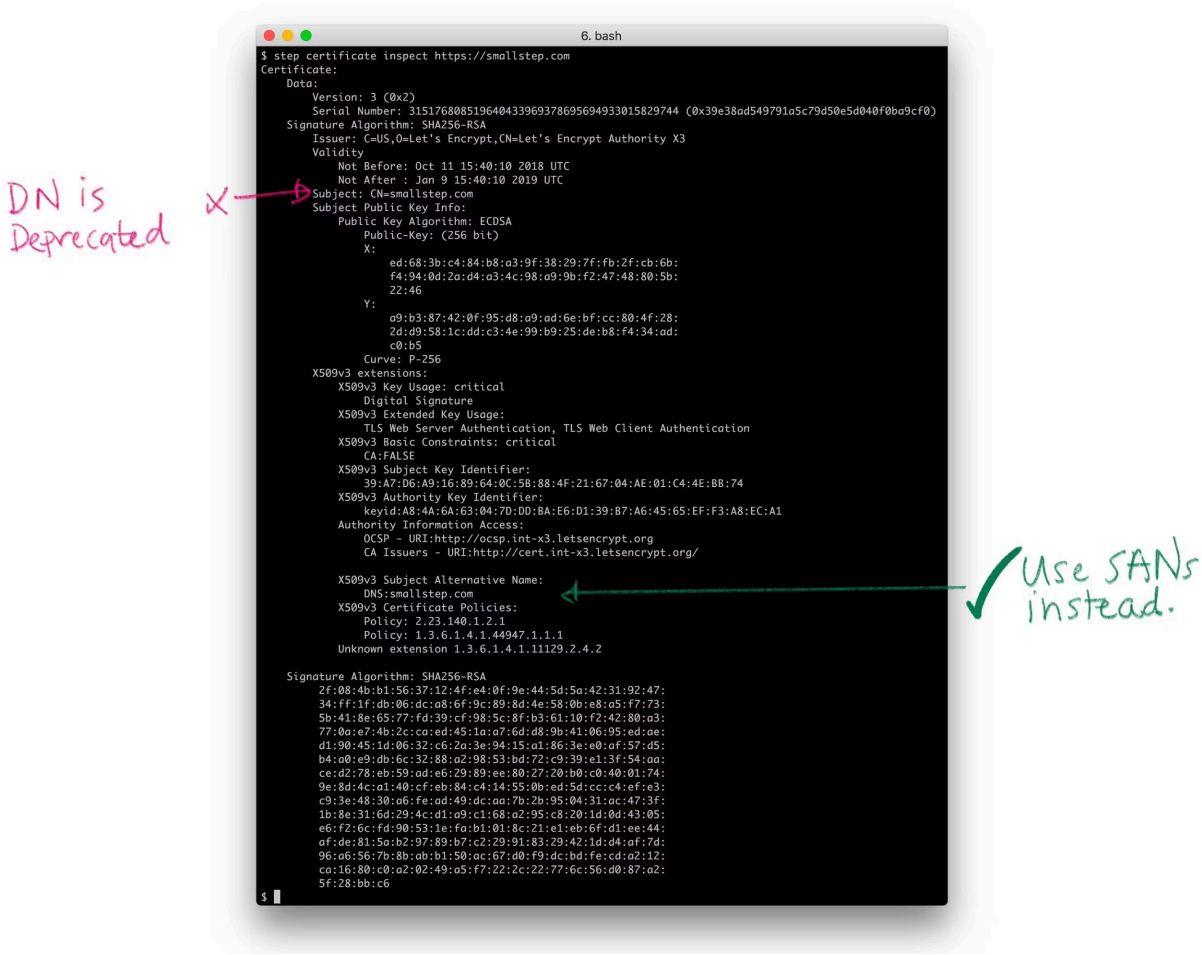
Like much of PKI this simple process is deceptively intricate. Hidden in the details are the two hardest problems in computer science: cache invalidation and naming things. Still, it's all easy enough to reason about once you understand what's going on.

Naming things

Historically, X.509 used X.500 *distinguished names* (DNs) to name the subject of a certificate (a subscriber). A DN includes a *common name* (for me, that'd be "Mike Malone"). It can also include a *locality*, *country*, *organization*, *organizational unit*, and a whole bunch of other irrelevant crap (recall that this stuff was originally meant for a digital phone book). No one understands distinguished names. They don't really make sense for the web. Avoid them. If you do use them, keep them simple. You don't have to use every field. In fact, you *shouldn't*. A common name is probably all you need, and perhaps an organization name if you're a thrill seeker.

PKIX originally specified that the DNS hostname of a website should be bound in the the DN *common name*. More recently, the CAB Forum has deprecated this practice and made the entire DN optional (see sections 7.1.4.2 of the Baseline Requirements). Instead, the modern best practices is to leverage the subject alternative name (SAN) X.509 extension to bind a name in a certificate.

There are four sorts of SANs in common use, all of which bind names that are broadly used and understood: domain names (DNS), email addresses, IP addresses, and URIs. These are already supposed to be unique in the contexts we're interested in, and they map pretty well to the things we're interested in identifying: email addresses for people, domain names and IP addresses for machines and code, URIs if you want to get fancy. Use SANs.



Note also that Web PKI allows for multiple names to be bound in a certificate and allows for wildcards in names. A certificate can have multiple SANs, and can have SANs like `*.smallstep.com`. This is useful for websites that respond to multiple names (e.g., `smallstep.com` and `www.smallstep.com`).

Generating key pairs

Once we've got a name we need to generate a key pair before we can create a certificate. Recall that the security of a PKI depends critically on a simple invariant: that the only entity that knows a given private key is the subscriber named in the corresponding certificate. To be sure that this invariant holds, best practice is to have the subscriber generate its own key pair so it's the only thing that *ever* knows it. Definitely avoid transmitting a private key across the network.

You'll need to decide what type of key you want to use. That's another post entirely, but here's some quick guidance (as of May 2023). There's a slow but ongoing transition from RSA to elliptic curve keys (ECDSA or EdDSA). If you decide to use RSA keys make them at least 2048 bits, and don't bother with anything bigger than 4096 bits. And use RSA-PSS, not RSA PKCS#1. If you use ECDSA, the P-256 curve is probably best (`secp256k1` or `prime256v1` in openssl)... unless you're worried about the NSA in

which case you may opt to use something fancier like EdDSA with Curve25519 (though support for these keys is not great).

Here's an example of generating a elliptic curve P-256 key pair using `openssl`:

```
openssl ecparam -name prime256v1 -genkey -out k.prv
openssl ec -in k.prv -pubout -out k.pub
```

Here's an example of generating the same sort of key pair using `step`:

```
step crypto keypair --kty EC --curve P-256 k.pub k.prv
```

You can also do this programmatically, and never let your private keys touch disk.

Choose your poison.

Issuance

Once a subscriber has a name and key pair the next step is to obtain a leaf certificate from a CA. The CA is going to want to authenticate (prove) two things:

- The public key to be bound in the certificate is the subscriber's public key (i.e., the subscriber knows the corresponding private key)
- The name to be bound in the certificate is the subscriber's name

The former is typically achieved via a simple technical mechanism: a certificate signing request. The latter is harder. Abstractly, the process is called identity proofing or registration.

Certificate signing requests

To request a certificate a subscriber submits a *certificate signing request* (CSR) to a certificate authority. The CSR is another ASN.1 structure, defined by PKCS#10.

Like a certificate, a CSR is a data structure that contains a public key, a name, and a signature. It's self-signed using the private key that corresponds to the public key in the CSR. This signature proves that whatever created the CSR knows the private key. It also allows the CSR to be copy-pasted and shunted around without the possibility of modification by some interloper.

CSRs include lots of options for specifying certificate details. In practice most of this stuff is ignored by CAs. Instead most CAs use a template or provide an administrative interface to collect this information.

You can generate a key pair and create a CSR using `step` in one command like so:

```
step certificate create --csr test.smallstep.com test.csr test.key
```

OpenSSL is super powerful, but a lot more annoying.

Identity proofing

Once a CA receives a CSR and verifies its signature the next thing it needs to do is figure out whether the name to be bound in the certificate is actually the correct name of the subscriber. This is tricky. The whole point of certificates is to allow RPs to authenticate subscribers, but how is the CA supposed to authenticate the subscriber before a certificate's been issued?

The answer is: it depends. For Web PKI there are three kinds of certificates and the biggest differences are how they identify subscribers and the sort of identity proofing that's employed. They are: domain validation (DV), organization validation (OV), and extended validation (EV) certificates.

DV certificates bind a DNS name and are issued based on proof of control over a domain name. Proofing typically proceeds via a simple ceremony like sending a confirmation email to the administrative contact listed in WHOIS records. The ACME protocol, originally developed and used by Let's Encrypt, improves this process with better automation: instead of using email verification an ACME CA issues a challenge that the subscriber must complete to prove it controls a domain. The challenge portion of the ACME specification is an extension point, but common challenges include serving a random number at a given URL (the HTTP challenge) and placing a random number in a DNS TXT record (the DNS challenge).

OV and EV certificates build on DV certificates and include the name and location of the organization that owns the bound domain name. They connect a certificate not just to a domain name, but to the legal entity that controls it. The verification process for OV certificates is not consistent across CAs. To address this, CAB Forum introduced EV certificates. They include the same basic information but mandate strict verification (identity proofing) requirements. The EV process can take days or weeks and can include public records searches and attestations (on paper) signed by corporate officers (with pens). And at the end of the day, web browsers don't prominently differentiate EV certificates in any way. So, EV certificates aren't widely leveraged by Web PKI relying parties.

Essentially every Web PKI RP only requires DV level assurance, based on "proof" of control of a domain. It's important to consider what, precisely, a DV certificate *actually* proves. It's *supposed* to prove that the entity requesting the certificate owns the relevant domain. It *actually* proves that, at some point in time, the entity requesting the certificate was able to read an email *or* configure DNS *or* serve a secret via HTTP. The underlying security of DNS, email, and BGP that these processes rely on is not great. Attacks against this infrastructure have occurred with the intent to obtain fraudulent certificates.

For internal PKI you can use any process you want for identity proofing. You can probably do better than relying on DNS or email the way Web PKI does. This might seem hard at first, but it's really not. You can leverage existing trusted infrastructure: whatever you use to provision your stuff should also be able to measure and attest to the identity of whatever's being provisioned. If you trust Chef or Puppet or Ansible or Kubernetes to put code on servers, you can trust them for identity attestations. If you're using raw AMIs on AWS you can use instance identity documents (GCP and Azure have similar functionality).

Your provisioning infrastructure *must* have some notion of identity in order to put the right code in the right place and start things up. And you *must* trust it. You can leverage this knowledge and trust to configure RP trust stores and bootstrap subscribers into your internal PKI. All you need to do is come up with some way for your provisioning infrastructure to tell your CA the identity of whatever's starting up. Incidentally, this is precisely the gap step certificates was designed to fill.

Expiration

Certificates expire... usually. This isn't a strict requirement, per se, but it's almost always true. Including an expiration in a certificate is important because certificate use is disaggregated: in general there's no central authority that's interrogated when a certificate is verified by an RP. Without an expiration date, certificates would be trusted forever. A rule of thumb for security is that, as we approach forever, the probability of a credential becoming compromised approaches 100%. Thus, certificates expire.

In particular, X.509 certificates include a validity period: an *issued at* time, a *not before* time, and a *not after* time. Time marches forward, eventually passes the *not after* time, and the certificate dies. This seemingly innocuous inevitability has a couple important subtleties.

First, there's nothing stopping a particular RP from accepting an expired certificate by mistake (or bad design). Again, certificate use is disaggregated. It's up to each RP to check whether a certificate has expired, and sometimes they mess up. This might happen if your code depends on a system clock that isn't properly synchronized. A common scenario is a system whose clock is reset to the unix epoch that doesn't trust any certificates because it thinks it's January 1, 1970 — well before the *not before* time on any recently issued certificate. So make sure your clocks are synchronized!

On the subscriber side, private key material needs to be dealt with properly after certificate expiration. If a key pair was used for signing/authentication (e.g., with TLS) you'll want to delete the private key once it's no longer needed. Keeping a signing key around is an unnecessary security risk: it's no good for anything but fraudulent signatures. However, if your key pair was used for encryption the situation is different. You'll need to keep the private key around as long as there's still data encrypted under the key. If you've ever been told not to use the same key pair for signing and encryption, this is the main reason. Using the same key for signing and encryption makes it impossible to implement key lifecycle management best practices when a private key is no longer needed for signing: it forces you to keep signing keys around longer than necessary if it's still needed to decrypt stuff.

Renewal

If you're still using a certificate that's about to expire you're going to want to renew it before that happens. There's actually no standard renewal process for Web PKI -- there's no formal way to extend the validity period on a certificate. Instead you just replace the expiring certificate with a new one. So the renewal process is the same as

the issuance process: generate and submit a CSR and fulfill any identity proofing obligations.

For internal PKI we can do better. The easiest thing to do is to use your old certificate with a protocol like mutual TLS to renew. The CA can authenticate the client certificate presented by the subscriber, re-sign it with an extended expiry, and return the new certificate in response. This makes automated renewal very easy and still forces subscribers to periodically check in with a central authority. You can use this checkin process to easily build monitoring and revocation facilities.

In either case the hardest part is simply remembering to renew your certificates before they expire. Pretty much everyone who manages certificates for a public website has had one expire unexpectedly, producing an error like this. My best advice here is: if something hurts, do it more. Use short lived certificates. That will force you to improve your processes and automate this problem away. Let's Encrypt makes automation easy and issues 90 day certificates, which is pretty good for Web PKI. For internal PKI you should probably go even shorter: twenty-four hours or less. There are some implementation challenges -- hitless certificate rotation can be a bit tricky -- but it's worth the effort.

Quick tip, you can use `step` to check the expiry time on a certificate from the command line:

```
step certificate inspect cert.pem --format json | jq .validity.end
step certificate inspect https://smallstep.com --format json | jq .valid
```

It's a little thing, but if you combine this with a DNS zone transfer in a little bash script you can get decent monitoring around certificate expiration for all your domains to help catch issues before they become outages.

Revocation

If a private key is compromised or a certificate's simply no longer needed you might want to revoke it. That is, you might want to actively mark it as invalid so that it stops being trusted by RPs immediately, even before it expires. Revoking X.509 certificates is a

big mess. Like expiration, the onus is on RPs to enforce revocations. Unlike expiration, the revocation status can't be encoded in the certificate. The RP has to determine the certificate's revocation status via some out-of-band process. Unless explicitly configured, most Web PKI TLS RPs don't bother. In other words, by default, most TLS implementations will happily accept revoked certificates.

For internal PKI the trend is towards accepting this reality and using *passive revocation*. That is, issuing certificates that expire quickly enough that revocation isn't necessary. If you want to "revoke" a certificate you simply disallow renewal and wait for it to expire. For this to work you need to use short-lived certificates. How short? That depends on your threat model (that's how security professionals say ¯\(ツ)/¯). Twenty-four hours is pretty typical, but so are much shorter expirations like five minutes. There are obvious challenges around scalability and availability if you push lifetimes too short: every renewal requires interaction with an online CA, so your CA infrastructure better be scalable and highly available. As you decrease certificate lifetime, remember to keep all your clocks in sync or you're gonna have a bad time.

For the web and other scenarios where passive revocation won't work, the first thing you should do is stop and reconsider passive revocation. If you *really* must have revocation you have two options:

- Certificate Revocation Lists (CRLs)
- Online Certificate Signing Protocol (OCSP)

CRLs are defined along with a million other things in RFC 5280. They're simply a signed list of serial numbers identifying revoked certificates. The list is served from a *CRL distribution point*: a URL that's included in the certificate. The expectation is that relying parties will download this list and interrogate it for revocation status whenever they verify a certificate. There are some obvious problems here: CRLs can be big, and distribution points can go down. If RPs check CRLs at all they'll heavily cache the response from the distribution point and only sync periodically. On the web CRLs are often cached for days. If it's going to take that long for CRLs to propagate you might as well just use passive revocation. It's also common for RPs to *fail open* -- to accept a certificate if the the CRL distribution point is down. This can be a security issue: you can trick an RP into accepting a revoked certificate by mounting a denial of service attack against the CRL distribution point.

For what it's worth, even if you're using CRLs you should consider using short-lived certificates to keep CRL size down. The CRL only needs to include serial numbers for certificates that are revoked *and* haven't yet expired. If your certs have shorter lifetimes, your CRLs will be shorter.

If you don't like CRL your other option is OCSP, which allows RPs to query an *OCSP responder* with a certificate serial number to obtain the revocation status of a particular certificate. Like the CRL distribution point, the OCSP responder URL is included in the certificate. OCSP sounds sweet (and obvious), but it has its own problems. It raises serious privacy issues for Web PKI: the OCSP responder can see what sites I'm visiting based on the certificate status checks I've submitted. It also adds overhead to every TLS connection: an additional request has to be made to check revocation status. Like CRL, many RPs (including browsers) fail open and assume a certificate is valid if the OCSP responder is down or returns an error.

OCSP stapling is a variant of OCSP that's supposed to fix these issues. Instead of the relying party hitting the OCSP responder the subscriber that owns the certificate does. The OCSP response is a signed attestation with a short expiry stating that the certificate is not revoked. The attestation is included in the TLS handshake ("stapled to" the certificate) between subscriber and RP. This provides the RP with a reasonably up-to-date revocation status without having to query the OCSP responder directly. The subscriber can use a signed OCSP response multiple times, until it expires. This reduces the load on the responder, mostly eliminates performance problems, and addresses the privacy issue with OCSP. However, all of this is a bit of a rube goldberg device. If subscribers are hitting some authority to obtain a short-lived signed attestation saying that a certificate hasn't expired why not cut out the middleman: just use short-lived certificates.

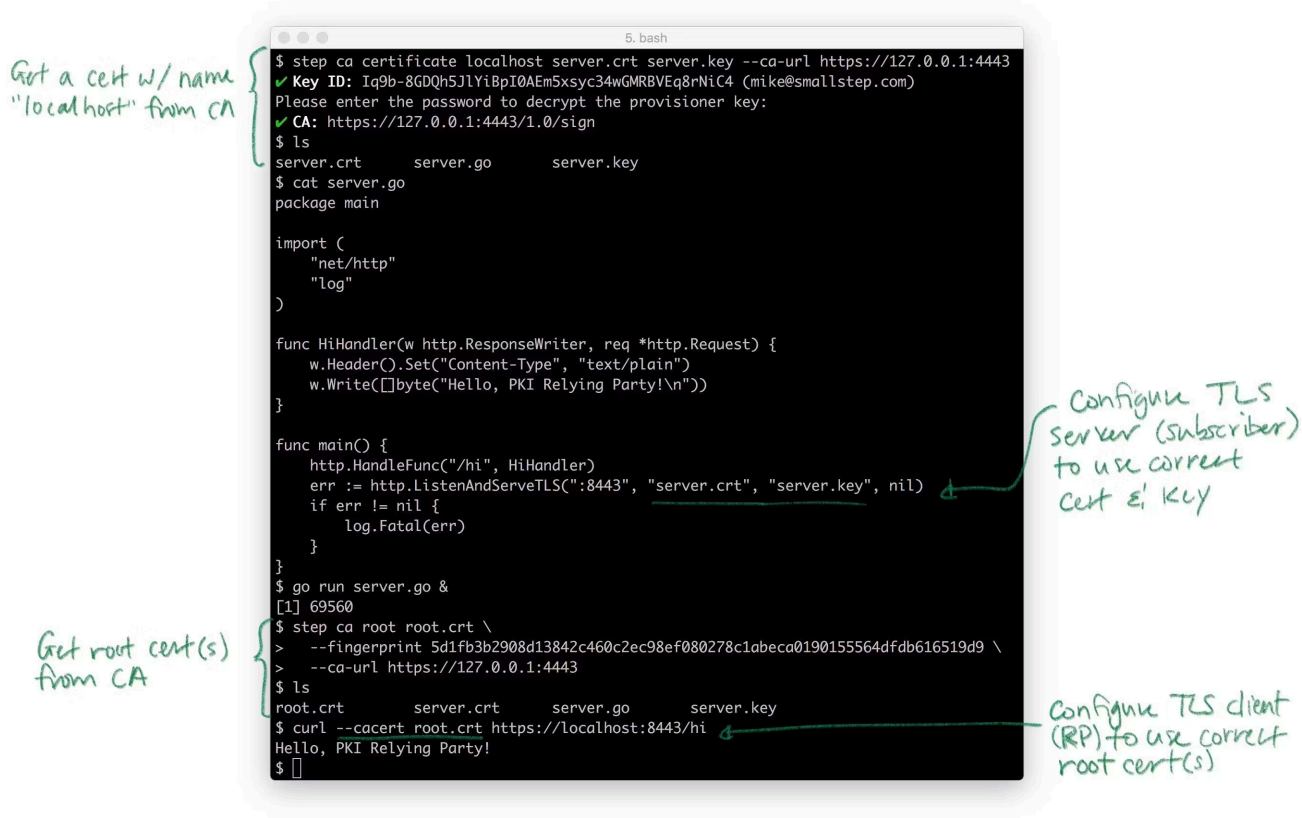
USING CERTIFICATES

With all of this background out of the way, actually *using* certificates is really easy. We'll demonstrate with TLS, but most other uses are pretty similar.

- To configure a PKI relying party you tell it which root certificates to use
- To configure a PKI subscriber you tell it which certificate and private key to use (or tell it how to generate its own key pair and exchange a CSR for a certificate itself)

It's pretty common for one entity (code, device, server, etc) to be both an RP and a subscriber. Such entities will need to be configured with the root certificate(s) and a certificate and private key. Finally, for Web PKI the right root certificates are generally trusted by default, so you can skip that part.

Here's a complete example demonstrating certificate issuance, root certificate distribution, and TLS client (RP) and server (subscriber) configuration:



Hopefully this illustrates how straightforward right and proper internal PKI and TLS can be. You don't need to use self-signed certificates or do dangerous things like disabling certificate path validation (passing `-k` to `curl`).

Pretty much every TLS client and server takes these same parameters. Almost all of them punt on the key and certificate lifecycle bit: they generally assume certificates magically appear on disk, are rotated, etc. That's the hard part. Again, if you need that, that's what step certificates does.

In Summary

Public key cryptography lets computers "see" across networks. If I have a public key, I can "see" you have the corresponding private key, but I can't use it myself. If I don't have your public key, certificates can help. Certificates bind public keys to the name of the owner of the corresponding private key. They're like driver's licenses for computers and code. Certificate authorities (CAs) sign certificates with their private keys, vouching for these bindings. They're like the DMV. If you're the only one who looks like you, and you show me a driver's license from a DMV I trust, I can figure out your name. If you're the only one who knows a private key, and you send me a certificate from a CA I trust, I can figure out your name.

In the real world most certificates are X.509 v3 certificates. They're defined using ASN.1 and usually serialized as PEM-encoded DER. The corresponding private keys are usually represented as PKCS#8 objects, also serialized as PEM-encoded DER. If you use Java or Microsoft you might run into PKCS#7 and PKCS#12 envelope formats. There's a lot of historical baggage here that can make this stuff pretty frustrating to work with, but it's more annoying than it is difficult.

Public key infrastructure is the umbrella term for all the stuff you need to build and agree on in order to use public keys effectively: names, key types, certificates, CAs, cron jobs, libraries, etc. Web PKI is the public PKI that's used by default by web browsers and pretty much everything else that uses TLS. Web PKI CAs are trusted but not trustworthy. Internal PKI is your own PKI that you build and run yourself. You want one because Web PKI wasn't designed for internal use cases, and because internal PKI is easier to automate, easier to scale, and gives you more control over a lot of important stuff like naming and certificate lifetime. Use Web PKI for public stuff. Use your own internal PKI for internal stuff (e.g., to use TLS to replace VPNs). Smallstep Certificate Manager makes building an internal PKI pretty easy.

To get a certificate you need to name stuff and generate keys. Use SANs for names: DNS SANs for code and machines, EMAIL SANs for people. Use URI SANs if these won't work. Key type is a big topic that's mostly unimportant: you can change key types and the actual crypto won't be the weakest link in your PKI. To get a certificate from a CA you

submit a CSR and prove your identity. Use short-lived certificates and passive revocation. Automate certificate renewal. Don't disable certificate path validation.

Remember: certificates and PKI bind names to public keys. The rest is just details.

Mike Malone has been working on making infrastructure security easy with Smallstep for six years as CEO and Founder. Prior to Smallstep, Mike was CTO at Betable. He is at heart a distributed systems enthusiast, making open source solutions that solve big problems in Production Identity and a published research author in the world of cybersecurity policy.

Production Identity | Step Certificates | Technical

Further Reading